EE4032

Spring semester 2023
Assignment report

Assignment	1
Assignment title	1-D Sum Case Study
Student name	Fionn Murray
Student ID number	18223451
Report submission date	14/04/2023

Complete Code

```
#include <cuda.h>
#include <stdio.h>
#include <math.h>

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#define RADIUS 128

#define N 10240000L
#define M 65537L

#define BLOCK_SIZE 128

// Soln2: Kernel code using shared memory
__global__ void kernelWithSharedMemory(int* inarray, int* outarray, long size, int radius, int modulus)
{
    __shared__ int tile[BLOCK_SIZE + 2 * RADIUS];

    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int lid = threadIdx.x + RADIUS;

    // Load data into shared memory
    if (gid < size)
    {
        tile[lid] = inarray[gid];
        if (threadIdx.x < RADIUS)
        {
            tile[lid - RADIUS] = inarray[gid - RADIUS];
            tile[lid + BLOCK_SIZE] = inarray[gid + BLOCK_SIZE];
        }
    }
    else
    {
        tile[lid] = 0;
    }

    __syncthreads();

    // Compute sum using shared memory
    int sum = 0;
    for (int i = lid - RADIUS; i <= lid + RADIUS; i++)
    {
        sum += tile[i];
    }

    // Store result to output array
    if (gid < size)
    {
        outarray[gid] = sum % modulus;
    }
}
```

```
void fillRandomData(int* a, long size)
{
    int i;
    for (i = 0; i < size; i++) {
        a[i] = rand() % M;
    }
}

void VerifyGPUOperation(int* inarray, int* outarray, long size)
{
    // Verification code
    for (int i = 0; i < size; i++)
    {
        int sum = 0;
        for (int j = i - RADIUS; j <= i + RADIUS; j++)
        {
            if (j >= 0 && j < size)
                sum += inarray[j];
        }
        int expected = sum % M;
        if (outarray[i] != expected)
        {
            printf("Verification failed at index %d: expected %d, got %d\n", i, expected, outarray[i]);
            return;
        }
    }
    printf("Verification successful!\n");
}

int main(void)
{
    // Input parameters
    int outputSize[] = { 10240,10240,10240,10240,10240,10240,10240, 1024000, 102400000 };
    int radius[] = { 3, 17, 128, 3, 3, 3, 3, 3, 3};
    int blockSize[] = { 128,128,128,128,512, 1024, 128, 128, 128 };
    int numTests = sizeof(outputSize) / sizeof(int);

    for (int i = 0; i < numTests; i++)
    {
        int size = outputSize[i];
        int rad = radius[i];
        int block = blockSize[i];

        printf("Test %d:\n", i + 1);
        printf("Output Array Size: %d\n", size);
        printf("Radius: %d\n", rad);
        printf("Block Size: %d\n", block);

        // Allocate memory for input and output arrays
        int* h_inarray = (int*)malloc(size * sizeof(int));
        int* h_outarray2 = (int*)malloc(size * sizeof(int));
        fillRandomData(h_inarray, size);

        int* d_inarray;
        int* d_outarray2;
```

```
        cudaMalloc((void**)&d_inarray, size * sizeof(int));
        cudaMalloc((void**)&d_outarray2, size * sizeof(int));
        cudaMemcpy(d_inarray, h_inarray, size * sizeof(int), cudaMemcpyHostToDevice);

        // Create CUDA events for measuring execution time
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);


        // Launch kernel for Soln2 using shared memory
        cudaEventRecord(start, 0);
        int numBlocks2 = (size + BLOCK_SIZE - 1) / BLOCK_SIZE;
        int numThreads2 = BLOCK_SIZE;
        kernelWithSharedMemory << <numBlocks2, numThreads2 >> > (d_inarray, d_outarray2, size, rad, M);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        float elapsedTimeWithSharedMemory;
        cudaEventElapsedTime(&elapsedTimeWithSharedMemory, start, stop);
        printf("Kernel Execution Time (with shared memory): %f ms\n", elapsedTimeWithSharedMemory);
        cudaDeviceSynchronize();
        cudaMemcpy(h_outarray2, d_outarray2, size * sizeof(int), cudaMemcpyDeviceToHost);


        // Copy results from device to host
        cudaMemcpy(h_outarray2, d_outarray2, size * sizeof(int), cudaMemcpyDeviceToHost);


        // Verify Soln2
        printf("Verifying Soln2...\n");
        VerifyGPUOperation(h_inarray, h_outarray2, size);

        // Free allocated memory
        free(h_inarray);
        free(h_outarray2);
        cudaFree(d_inarray);
        cudaFree(d_outarray2);

        printf("\n");
    }

    printf("Finished with success\n");
    exit(EXIT_SUCCESS);
}
```

Explanation Of Kernel Function

```
15  // Soln2: Kernel code using shared memory
16  __global__ void kernelWithSharedMemory(int* inarray, int* outarray, long size, int radius, int modulus)
17  {
18      __shared__ int tile[BLOCK_SIZE + 2 * RADIUS];
19
20      int gid = blockIdx.x * blockDim.x + threadIdx.x;
21      int lid = threadIdx.x + RADIUS;
22
23      // Load data into shared memory
24      if (gid < size)
25      {
26          tile[lid] = inarray[gid];
27          if (threadIdx.x < RADIUS)
28          {
29              tile[lid - RADIUS] = inarray[gid - RADIUS];
30              tile[lid + BLOCK_SIZE] = inarray[gid + BLOCK_SIZE];
31          }
32      }
33      else
34      {
35          tile[lid] = 0;
36      }
37
38      __syncthreads();
39
40      // Compute sum using shared memory
41      int sum = 0;
42      for (int i = lid - RADIUS; i <= lid + RADIUS; i++)
43      {
44          sum += tile[i];
45      }
46
47      // Store result to output array
48      if (gid < size)
49      {
50          outarray[gid] = sum % modulus;
51      }
52  }
```

Figure 1, Kernel Function

The kernel function uses the inputs ‘inarray’ and ‘outarray’ as pointers to input and output arrays, ‘size’ as the size of the input array, ‘radius’ as the neighborhood radius, and modulus as the value for modulo operation. The function uses global and local thread indices to compute the neighborhood sum for each element in the input array. The second line of the function declares a shared memory array named tile with a size of BLOCK_SIZE + 2 * RADIUS. This array is intended to hold the data that will be processed by each thread in the block.

To ensure the thread processes the correct element in the input and output arrays, the third line then computes the global thread index gid using the blockIdx.x and blockDim.x variables. The

threadIdx.x variable represents the index of the thread within the block, and the RADIUS constant is added to it to account for the padding that will be added to the shared memory array.

The next section loads the inputted data into shared memory. If the thread's global index is less than size, the tile array is populated with the corresponding element of the inarray.

If the thread is in the first RADIUS threads of the block, it loads in the previous RADIUS elements from inarray. Similarly, if the thread is in the last RADIUS threads of the block, it loads in the next RADIUS elements from inarray.

If the thread's global index is greater than or equal to size, the corresponding element in the shared memory tile array is set to zero.

The __syncthreads() function is called after the data has been loaded into shared memory to ensure that all threads in the block have finished loading their data before any computation is done.

The next section of the code computes the sum of the values using the data stored in tile. The variable sum is initialized to zero, and a loop is used to iterate over the elements in tile that correspond to the sliding window.

Finally, the last section of the code writes the computed result to the output array outarray if the thread's global index is less than size. The result is computed by taking the modulus of sum with respect to the modulus parameter.

```
Output Array Size: 10240
Radius: 3
Block Size: 128
Kernel Execution Time (with shared memory): 0.115712 ms
Verifying Soln2...
Verification successful!

Test 2:
Output Array Size: 1024000
Radius: 17
Block Size: 512
Kernel Execution Time (with shared memory): 2.131968 ms
Verifying Soln2...
Verification successful!

Test 3:
Output Array Size: 102400000
Radius: 128
Block Size: 1024
Kernel Execution Time (with shared memory): 209.136642 ms
Verifying Soln2...
Verification successful!

Finished with success
```

Figure 2, Segment of Code output debug console

Plot Of Test Case Execution Times

Similar to the first solution not using shared memory, the primary parameter to have an impact on kernel execution time is the chosen size of the output array. However, more interestingly in these results is an evident increase in kernel execution time consistency.

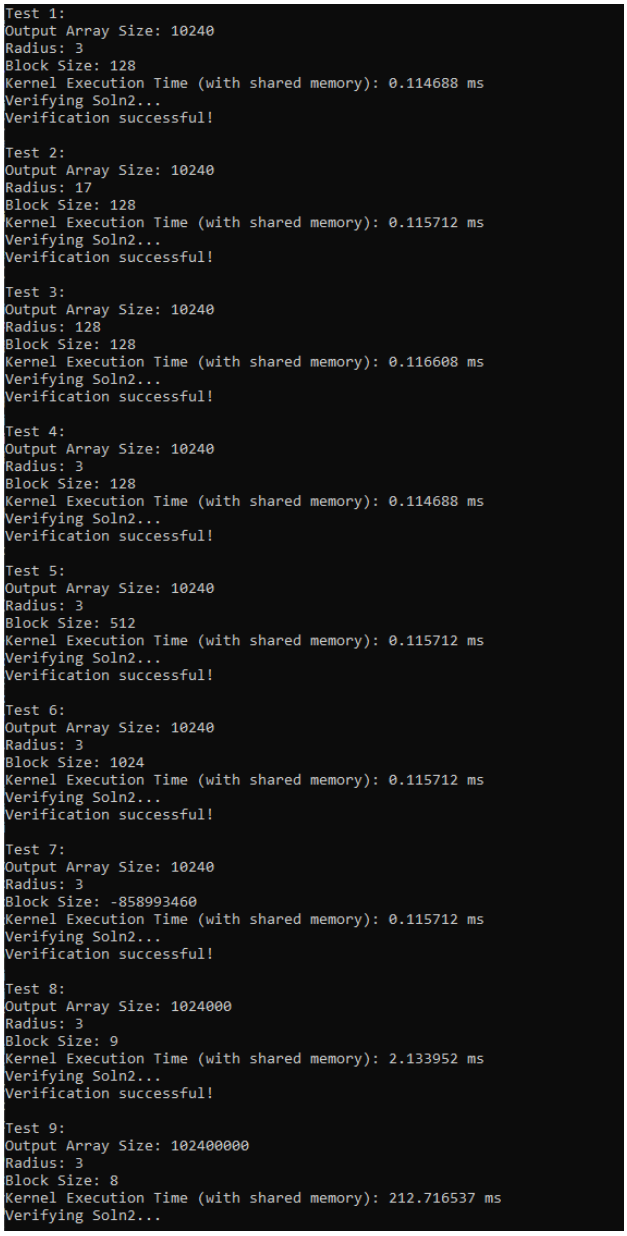

Figure 3, Initial Test

This test first tests the impact of increasing the radius, then the block size, and finally the output array. The increase in radius size has a small impact on the kernel execution time, while the large change in output array size has a much more significant impact on the execution time. The plots below reflect this.

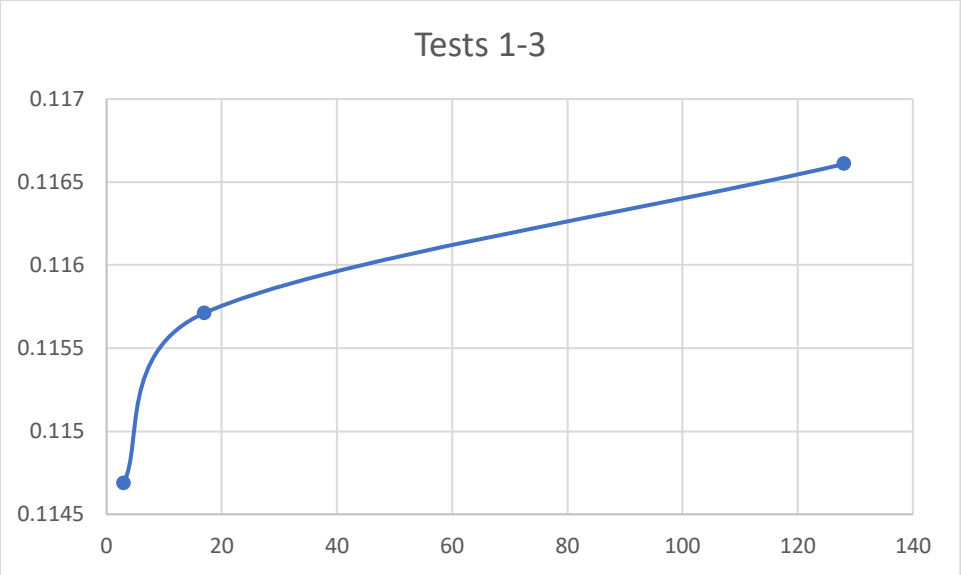

Figure 4, Varying Radius Length

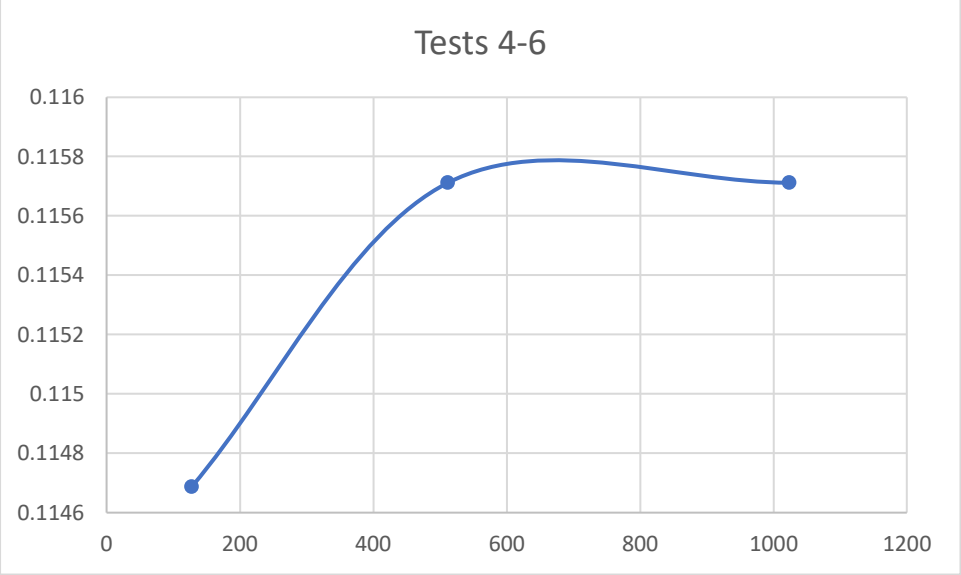

Figure 5, Varying Block Size

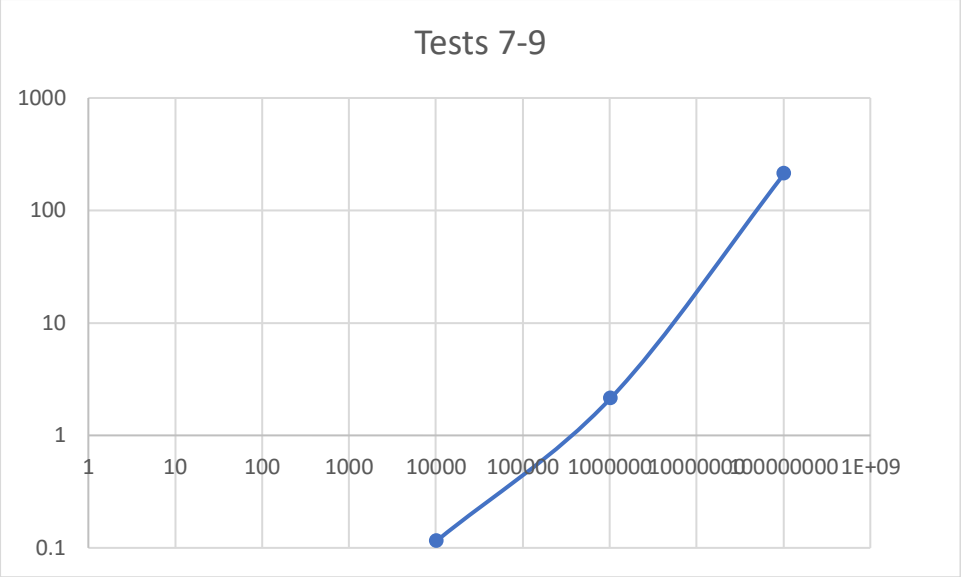

Figure 6, Varying Array Output Size